IBM® Workplace Forms™

Version 2.5

Working with XFDL: Tips and Tricks for Form Design

S229-1509-00

First Edition (December 2005)

This edition applies to version 2.5 of IBM® Workplace Forms™ .

Contents

Introduction

XFDL is an extremely flexible language. It offers "out of the box" support for a wide variety of forms, and includes most of the elements you would expect to see on a form, such as fields, check boxes, and so on. However, once you become familiar with the process of designing a simple form, you may find yourself wanting to add more complex behavior. For instance, you might want to create a group of check boxes that limits you to a single selection.

This document will cover some of the tips and tricks you can use to get more out of your forms. In the process, we'll teach you more about XFDL and how to use computes to achieve your goals.

Because IBM® Workplace Forms™ Designer offers limited support for writing complex computes, we're going to assume that you'll be working in the Code View window or in a text editor while adding these features to your forms.

Who Should Read this Document

This document is intended for readers who have a basic understanding of Workplace Forms, either through using Workplace Forms Designer or through writing forms in a text editor. You should be familiar with the basic XFDL syntax, and might want to keep the *XFDL Specification* handy as a reference.

Document Conventions

You should be familair with the following conventions when reading this document.

Line Breaks in Code Samples

It's often useful to use line breaks to format long computes in XFDL. This helps make them easier to read. For instance, consider the following compute:

```
<active compute="
    SmokerCheck.value == 'Yes'
    ? 'on'
    : 'off'"></active>
```

This is a simple if/then/else test, formatted so that each part of the test is on separate line. It reads, if the value of SmokerCheck is Yes (line 2), then active is on (line 3), else active is off (line 4). You can see with this small example how breaking the compute into multiple lines can make it easier to read.

However, if you want Workplace Forms software to preserve those line breaks, you have to add manual line returns to each line.

In XML (and therefore XFDL), a line return is written as the following set of symbols:

```
&#xA;
```

This document will include these hard returns, so that you can copy and paste the sample code into your forms without worrying about losing the line breaks. Also, for the sake of readability, we will move all of the line breaks to the right margin of the page, as shown:

```
<active compute="                                                &#xA;
   SmokerCheck.value == 'Yes'                                    &#xA;
   ? 'on'                                                        &#xA;
   : 'off'"></active>
```

Be sure to include the line break symbols when you copy and paste any samples into your forms.

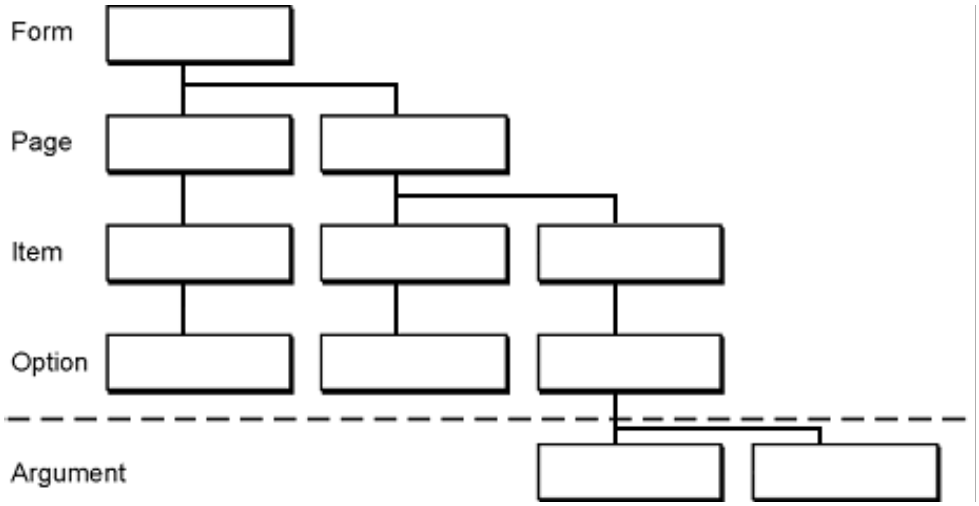
Getting Started with Computes

Most of the advanced features you'll want to add to a form require the use of computes. You can think of computes as the internal logic of a form — they can copy values, make decisions, and even change the appearance of the form.

However, before you can use computes properly, you need a basic understanding of a form's node structure, and how to reference different parts of a form.

Understanding a Form's Node Structure

When a form is loaded into memory by a Workplace Forms product, it is converted into a *node structure*. The node structure is basically a tree structure, similar to a family tree, with two main differences. First, instead of family members, each node represents an element in the form, and second, the tree is upside down, as shown:



As you can see from the diagram, each level of the node structure represents a different part of the form:

- **Form** — This is the highest level of the tree. It always consists of just one node: the form node. This node corresponds to the <XFDL> tag in a form, and all other nodes are descendants of it.
- **Page** — This level always contains a minimum of two page nodes. The first is a special node that represents the global page, which is used to store default settings that apply to the whole form. The second node represents the first page in the form. As you create new pages in your form, a new page node is added to represents each additional page.

- **Item** — Item nodes are descendants of page nodes, and represent the items on that page. Items are always descendants of a page node, and there can be an unlimited number of items within a page.
- **Option** — Option nodes are the children of item nodes, and represent the options that are set for a particular item. Options are always descendants of an item node, and there can be an unlimited number of options within an item.
- **Argument** — These nodes store settings for option nodes. Argument nodes can nest below an option node. This means that argument nodes can themselves be the parents of other argument nodes. There can be an unlimited number of sub-levels in the argument level.

As you add elements to the form, the node structure grows to the right to reflect each new page, item, and so on.

Given the size of most business forms, it would be a large job to actually map out the node structure for an entire form. Fortunately, you never really have to do that. It's enough to understand the rules for creating the structure, because it's those rules that tell you how to make references to particular parts of your form.

Understanding References

Before you can write complex computes, you have to know how to refer to particular parts of the form. For example, you might want to write a compute that will change the value of a particular field. But first you have to know how to identify that particular value.

Basic References

XFDL uses references to refer to particular elements in a form, much like you and I would use names to refer to each other. You can think of a reference like a map — it's a short set of directions that tell you how to find the element you're looking for.

XFDL references are based on the node structure, and take a top down format. By this, I mean that a references begins at the top the node structure and works its way down to the element you want. To do this, we use a dotted notation that follows this format:

```
<page name>.<item name>.<option name>
```

In this case, the page name is the SID of the page, and the item name is the SID of the item. Options are always named for their type. For example, the bgcolor option is named "bgcolor". As you can see, each name is separated by a period. That's why we call it a dotted notation — elements are separated by dots.

To write an actual reference, you have to understand the structure of your form. For instance, suppose you had a form with one page named "Page1", two fields on that page named "Field1" and "Field2", and that each field had a value option. If you wanted to refer to the value of the second field, you would use the following reference:

```
Page1.Field1.value
```

Shortening References

When you use a reference, you have to place the reference in a compute in your form. For example, you might use a reference in the value of field to refer to the value of a different field, as shown:

```
<field sid="Field1">
   <value compute="Page1.Field2.value"></value>
</field>
```

In this case, the value of Field1 is referencing the value of Field2 (which simply copies the value of Field2 into the value of Field1).

In cases like this, you can often shorten your reference by following these rules:

1. If the element you are referencing is on the same page as the element that contains the reference, then you do not need to include the page in your reference. For example, you might use:

   ```
   Field1.value
   ```

2. If the element that you are referencing is in the same item as the element that contains the reference, you do not need to include the item in your reference. For example, you might use:

   ```
   value
   ```

3. You must always include the option in your references.

Referencing Arguments

You can also reference arguments. These are individual settings within an option that are contained in the <ae> elements. For example, you will often have a color option that has three arguments or settings: the red, the green, and the blue.

To reference arguments, you use an index notation, as shown:

```
<page name>.<item name>.<option name>[index]
```

The index is added in brackets to the end of the reference. The index itself is a number, beginning with zero and counting up, as shown:

```
<color>
   <ae>red</ae>        <-- 0
   <ae>green</ae>      <-- 1
   <ae>blue</ae>       <-- 2
</color>
```

So refer to the red, you would use [0], to refer to the green you would use [1], and to refer to the blue you would use [2]. If the color option was part of Field1 on Page1, you would use the following full reference to refer to the blue color:

```
Page1.Field1.color[2]
```

Arguments can also be nested in an option. For example, you might have <ae> elements that contain additional <ae> elements. This is common with the itemlocation option. This sort of nesting creates an array of arguments, as shown:

```
<itemlocation>
   <ae>                      <--- [0] at level 1
      <ae>absolute</ae>         <------- [0] at level 2
      <ae>220</ae>              <------- [1] at level 2
      <ae>60</ae>               <------- [2] at level 2
   </ae>
   <ae>                      <--- [1] at level 1
      <ae>extent</ae>           <------- [0] at level 2
      <ae>100</ae>              <------- [1] at level 2
      <ae>40</ae>               <------- [2] at level 2
   </ae>
</itemlocation>
```

As you can see, each <ae> at level one contains its own set of <ae> elements at level two, and the numbering resets within each group.

To refer to an argument in an array, you can use more than one index. The first index will refer to the first level of the array, while the second index will refer to the second level of the array, and so on, as shown:

```
<page name>.<item name>.<option name>[index level 1][index level
2]...
```

For example, to refer to the value of “40” in the previous example, you would use the following reference:

```
Page1.Field1.itemlocation[1][2]
```

References and Namespace

If you are referring to an option that is not part of the core XFDL namespace, you must add the namespace prefix to the custom option reference:

```
PAGE1.FIELD1.custom:myoption
```

Types of Computes

In general, there are four types of computes that you will use:

- **Reference Computes** — References computes simply copy data from one element to another.
- **Mathematical Computes** — Mathematical computes perform basic math, and are useful for making the type of calculations you would find in a spreadsheet.
- **Conditional Computes** — Conditional computes actually make decisions based on data in the form.
- **Functions** — Functions are computes that resemble function call in programming languages. You give them some input, and they do some work and give you an output. In general, you use functions to perform calculations that would be too complicated to express as other types of computes.

In many cases, you'll actually constuct computes by combining these types. For instance, mathematical and conditional computes will almost always contian reference computes.

The four types of computes are explained in more detail in the following sections.

Reference Computes

You can use a simple reference to copy data from one element of your form to another. For example, consider the following form:

```
<field sid="Field1">
   <value compute="Page1.Field2.value"></value>
</field>
<field sid="Field2">
   <value>Hello</value>
</field>
```

In this case, the value option for Field1 has a compute that is simply a reference to the value option for Field2. When you reference an element directly in this way, you are saying that you simply want to copy the value. In this case, "Hello" would be copied from Field2 and set as the value for Field1.

Reference computes like this are best used in item that the use can't change. For example, you might have a two page form with an address block on each page. When the user types their address on the first page, you might use reference computes to automatically copy that address to the second page. In this case, you may also want to make the second address block read only, so that the user cannot change the copied values. The computes themselves will not prevent the user from typing new values into the address block on the second page.

Mathematical Computes

You can use mathematical computes to perform simple math, similar to the calculations you can perform in a spreadsheet. Mathematical computes combine references with mathematical symbols to express simple formulas.

For example, consider the following form:

```
<field sid="Field1">
   <value compute="Field2.value + Field3.value"></value>
</field>
<field sid="Field2">
   <value>5</value>
</field>
<field sid="Field2">
   <value>10</value>
</field>
```

In this form, the value of Field1 is set by a mathematical compute. This compute contains two references and an addition symbol. Each reference copies a value from another field in the form — a 5 from Field2 and a 10 from Field3. This produces the following expression:

```
5 + 10
```

Of course, this equals 15. Once the compute is evaluated, the final value is then set into the option. So in this case, the value of Field1 is set to 15.

Mathematical Operators

When writing mathematical computes, you can use the following operators:

Type of Operator	Symbol	Operation
Additive	+ - (minus) +.	addition subtraction concatenation
Multiplicative	* / %	multiplication division modulus (returns remainder)
Exponentiation	^	exponential
Unary Minus	- (minus)	take negative

Conditional Computes

You use conditional computes to make decisions. Conditional computes are often referred to as IF - THEN - ELSE expressions.

They are written symbolically as:

```
x ? y : z
```

You can read this as:

```
If x THEN y ELSE z
```

In other words:

IF condition x is true, THEN do y; otherwise do z.

For example, if you wanted a field on page 2 to copy information from a field on page 1 if a checkbox is selected, you would use the following compute to the field on page 2:

```
PAGE1.CHECK1.value == 'on' ? PAGE1.FIELD1.value : ""
```

You could read the above compute as:

If the checkbox is turned on, then place the value of PAGE1.FIELD1 into the value of PAGE2.FIELD2; otherwise, leave PAGE2.FIELD2 blank (empty quotes indicate an empty value).

In other words, if CHECK1 is not selected, the compute does nothing.

Conditional Operators

When writing conditional computes, you can any mathematical operators plus the following logic oriented operators:

Type of Operator	Symbol	Operation
Relational	>	greater than
	<	less than
	<=	less than or equal to
	>=	greater than or equal to
	==	equal to
	!=	not equal to
	Note: *The "less than" operator is not permitted as such; use escape sequence* **<**	
Logical	&&	AND
	and	AND
	||	OR
	or	OR
	!	NOT
	Note: *The reserved words* 'and' *and* 'or' *are case sensitive. Always use lower case.*	
Decision	x?y:z	assign the value of expression **y** to the result if expression **x** evaluates to true. Otherwise, assign the value of expression **z** to the result.

Precedence of Operations

When you use more than one operator in a compute, the operations are evaluated in the following order (from first to last):

- Membership

- Exponentiation
- Multiplicative and unary minus
- Additive
- Relational
- Logical NOT
- Logical AND
- Logical OR
- Conditional

Functions

You can think of functions as “block box” computes. Basically, you give them some input, and they do some work and give you some output. You don’t get to see the actual calculations they perform, but you do know what sort of output to expect.

Functions allows you to perform very complex operations that would be too difficult (or impossible) to express using other types of computes. For example, you can use the countWords function to count how many words the user entered into a particular field.

XFDL includes a set of standard functions that you can always use in your forms. These functions written as a single word that is often followed by one or more *parameters*. Parameters are just information that you need to give the function before it can do its work.

For example, the countWords function is written like this:

```
countWords(string)
```

In this case, the string is the group of words that you want to count. This will normally be a reference to a value that will be copied into the function. For example, if you wanted to count whatever the user typed into Field1, and display that value in Field2, you might have a form that looked like this:

```
<field sid="Field1">
   <value>This is what the user types.</value>
</field>
<field sid="Field2">
   <value compute="countWords(Field1.value)"></value>
</field>
```

In this case, the value of Field2 is set by a compute. This compute uses the countWords function, and uses a reference to copy the value of Field1 into that function. The function then returns the number of words (in this case, 6) and sets the value of Field2 to equal that result.

Restricted Characters in Computes

Because of some rules that apply to all XML languages, computes cannot contain the ampersand (&) and less than symbol (<). If you want to use these symbols, you must express them as *entity references*. These are basically markers that you can use in place of the symbols themselves.

The entity references for the ampersand and less than symbols are:

- & *as* &
- < *as* <

Commonly Used Computes

When designing forms, there are a number of common computes that are used to add functionality. This section discusses several of these computes, and provides complete examples of how you would use these computes in your form.

Mirroring Data

Mirroring allows you to copy data from one option to another. This means that one value is aways the mirror of another. Mirroring is accomplished by using simple computes to copy the data. Generally, this is most useful in forms with multiple pages that use common data.

For example, you may have a form that requires the user to provide their address on page one and page three. Instead of forcing them to re-type their address on each page, you can simply set up the form to copy their address from page one to page three.

When you copy information from one item to another like this, the user can still change the copy. For example, once the user's address was copied to page three, the user could then move on to page three and type in a completely different address. Furhtermore, when the user does this, the compute you added to create the mirror will be destroyed.

For this reason, you should always make you mirrors read-only. This will prevent the user from changing the data and ruining your computes.

There are two ways to mirror data. The first is using a simple reference, as shown:

```
<value compute="PAGE1.firstName_FIELD.value"></value>
```

The second method is dereferencing. When you dereference an item or option, you tell the compute to find the form element identified by a reference. In other words, if an option had a literal value that contained a reference, and you wanted to know what the value of the reference was, you would use a compute with a dereference to retrieve it. You would primarily use a dereferance to find the value of a cell in a popup or list. For example, the value of a list depends upon which list choice the user selected. Therefore, to discover the value of a list, you must first determine which cell was selected and then determine the value of the cell.

The following code sample shows a simple compute with a dereference:

```
<value compute="PAGE1.occupation_list.value->value"></value>
```

For additional help, see the example form *Mirror_Data.xfd*. Open the form in the Designer or a text editor and look at "FIELD2" to see a commented custom compute.

Pushing Data

Pushing data is the exact opposite of mirroring data. In mirroring, you copy (or pull) data from another location in the form to the location of the compute. Typically, pushing data takes information from its current location and 'pushes' it to another location in the form. For example, an field might have a compute that copies whatever the user enters into it to another field in the form.

This means that data is usually pushed as a reaction to a change in the form. For example, a form might contain a checkbox that users select to indicate they are married. When the checkbox is selected, a compute detects this event and changes the visibility of a set of items to **on**, allowing users to add information about their spouse.

You should contain computes that push data in a custom option so that a user entering data cannot accidentally overwrite the compute.

To push data, you need to use two functions:

- **toggle** — Watches for a change in the current item's value.
- **set** — Pushes the new data to another location in the form. You must specify the item and option to which this function pushes data.

The following code illustrates a conditional compute that pushes data from one field to another field.

```
<custom:push_value xfdl:compute="toggle(value) == '1' ?     &#xA;
   set('PAGE4.FIELD2.value', value) : ''"></custom:push_value>
```

Effectively, this compute reads: if the value of this field changes, then update the value of Field2; otherwise, do nothing.

Generally, you would have a similar compute in a custom option in Field2 that would update the original field if Field2's value changed, thus allowing the user to make a change in either location and have the form keep track of the data in both locations.

If you need to set the value of more than one item on a value change, use the "+" operator within the compute to add the additional operation.

```
<custom:push_values xfdl:compute = "toggle(value) == '1' ?  &#xA;
   set ('PAGE4.FIELD2.value', value) +                      &#xA;
   set('PAGE6.FIELD8.value',value) : '' "></custom:push_values>
```

For additional help, please see the example form *Pushing_Data.xfd*. Open the form in the Designer or a text editor and look at field "FIELD1" to see the custom compute.

Mutually Exclusive Checkboxes

Mutually exclusive checkboxes are checkboxes that act in a manner similar to radio buttons — you can only select one of them.

There are two ways to create mutually exclusive checkboxes:

- Have each check set the others.
- Track the last check that was selected.

Each Check Sets the Others

The simplest way to create mutually exclusive checkboxes to have each checkbox update all of the checkboxes when it is turned on.

This requires two functions:

- **toggle** — Watches for an event or a change of state in the current item's value going from “off” to "on", showing that the checkbox has been selected.
- **set** — Pushes the value of “off” to other checkboxes in the set. You must specify the option name and value that this function pushes data to; otherwise, set defaults to setting the item containing the compute.

The following code illustrates a simple case of two mutually exclusive check boxes.

Check1:

```
<custom:check1 xfdl:compute="toggle(value, 'off', 'on') ==  &#xA;
   '1' ? set('PAGE1.CHECK2.value', 'off') : ''"></custom:check1>
```

Check2:

```
<custom:check2 xfdl:compute="toggle(value, 'off', 'on') ==  &#xA;
   '1' ? set('PAGE1.CHECK1.value', 'off') : ''"></custom:check2>
```

The downside to this method is that with large groups of checks, the computes become lengthy and difficult to maintain when you add or remove checkboxes from the group.

Tracking the Last Check Selected

A more complex strategy is to keep track of the last check that was turned on. This method is more difficult to set up than the previous method, but far easier to maintain as if you need to add or remove checks from the group.

To keep track of which checkbox was last selected, you need to create a custom option in the first checkbox in the group, as shown:

```
<check sid="Check1">
   <value></value>
   <custom:lastSelected></custom:lastSelected>
</check>
```

In this case, you can use the lastSelected element to store a reference to the checkbox. By storing which checkbox was last selected, you will always know which checkbox to turn off when the user selects a different one.

You then need to add a compute to each checkbox that will do this work when the user selects it. This compute is also stored in a custom item, since you do not want the compute to affect any of the values in the checkbox itself. The following option shows a sample of the compute you would use:

```
<custom:check_set xfdl:compute = "toggle(value, 'off', 'on')&#xA;
   == '1' or toggle(value, '', 'on') == '1' ?              &#xA;
   set('Check1.custom:lastSelected->value', 'off') +       &#xA;
   set('Check1.custom:lastSelected',                       &#xA;
   getReference('', '', 'item') ) : ''"></custom:check_set>
```

This compute performs the following calculations:

1. The compute is only triggered if the value of the check box that contains it goes from **off** to **on**.
2. The compute then gets the reference to the last checkbox that was selected from the lastSelected option, and sets the value of that checkbox to **off**.
3. Finally, the compute sets the value of the lastSelected option to be a reference to the checkbox that was just selected.

Notice that we used the dereference operator to set the previously “on” checkbox to “off”. This is known as a dynamic reference.

The example form *MutuallyExChBoxes.xfd* demonstrates this methodology. Note that to add more checkboxes to the group, you need only copy the existing ones; you do not need to update their computes at all.

Duplicating Items and Pages

Sometimes you won't know how much information users need to enter into a form, and therefore don't know how large to make a form. Paper form designers know that users can write in the margins or on the back of a form, or even attach additional pages, but electronic form designers don't have that luxury. However, if you have a form that requires users to enter a variable amount of information, such as an accident report or a bibliography, you can create a small form that dynamically generates new pages or items as needed, instead of creating a large form with many pages or tables.

You can do this by using the duplicate function. The duplicate function can duplicate items and pages. If you use it to duplicate items, it automatically duplicates the item's options and arguments. If you use it to duplicate pages, it automatically duplicates every item and option on the page.

Note: *If you are not duplicating an entire item and its options, you should use the set function rather than the duplicate function.*

The duplicate function uses six parameters:

1. A reference to the form element that you want to duplicate.
2. The type of element referenced in parameter 1. For example, page or item.
3. A reference to a form element that will anchor the newly duplicated element (using relative positioning) in the form's build order.
4. The type of element referenced in parameter 3.
5. The relation of the new item to the anchor item. (Valid choices are append_child, after_sibling, and before_sibling.)
6. The name of the new form element.

The following sample shows how to write a simple duplicate function:

```
duplicate('TEMPLATE_PAGE', 'page', 'TEMPLATE_PAGE', 'page',
'before_sibling','PAGE')
```

Duplicate functions are usually contained within computes that are triggered by a user actions, and are therefore often partnered with set and toggle functions, as shown below:

```
<custom:on_activated xfdl:compute="toggle(activated, 'off', &#xA;
   'on') == '1' ? set('custom:page_count',                  &#xA;
   custom:page_count + '1') +. duplicate('TEMPLATE_PAGE',   &#xA;
   'page', 'TEMPLATE_PAGE', 'page', 'before_sibling', 'PAGE'&#xA;
   +. custom:page_count) +. set('PAGE' +. custom:page_count &#xA;
   +. '.pageNumber_LABEL.custom:myPage_num', page_count))   &#xA;
   : ''"></custom:on_activated>
```

Duplicating Pages

You can use the duplicate function to duplicate entire pages. This allows the user to add any number of pages to the form. In general, you do this by creating a template of the page you want to duplicate. This template is a page that the user can't access, and that is only used for copying. This means that the page normally contains no user data, so you can duplicate the page without having to worry about removing user data from it.

When duplicating a page, you usually want to place each page one after another so that the newest page has the highest page number. Your form will run faster if you place the new pages before a static (non-duplicated) page, such as your template page, rather than placing each new page after the last duplicated page. (This is because it requires more work to evaluate the SID of the last duplicated page than it does to evaluate a static page SID.)

The template page should contain buttons that allow the user to switch to the next page and the previous page, so that users can navigate through all the newly duplicated pages. However, you should also add a compute to the Next Page button that will disabled it if the next page is the template page.

Note: *You can simplify page navigation by using relative page tags in the code of the Next Page and Previous Page buttons, instead of setting their url option. The relative page tags are global.pagenext and global.pageprevious.*

Both the template page and the last page of the form usually contain an Add Page button. This button triggers the page duplication when it is clicked. You can add custom options to this button that store dynamic information, such as the number of pages that have been added to the form. You can then reference this information to display a running page total to the user.

Other methods of triggering page duplication include the automatic display of a message box. A message box is a popup message that the form displays that offers or requests additional information. In this case, your message box could read, "Add another page?" when the user leaves the last item on the last page. If users click "Yes", another page is automatically added to the form.

For additional help, see the example form *Page_Duplication.xfd*. Open the form in Workplace Forms Designer or a text editor and examine the page duplication button to see the computes.

Duplicating Items

If a form contains items that are arranged in a table, you may want to allow the user to dynamically add rows to the table. As with pages, this functionality requires the form to include a template. In this case, the template must be a row of items.

Usually, you place the template row on the same page as the table. It can be either the first row of the table, or be invisible so that it is only used for duplication purposes. Both techniques require about the same amount of work.

If you use the first row of the table as the template row, you must clear the value options of all the newly duplicated items so that they do not contain the user input from the first row. If you hide the template row, the value options of the duplicated items will be blank, but you must remember to set their visible options to on.

Setting up a template row of items requires a lot of configuration. You must configure the template row using relative positioning and relative item tags, so that duplicated items don't overlap the original items. Relative item tags include itemnext and itemprevious. Once you have configured each template item so that it is placed below or after the previous item in the build order, any new rows are automatically displayed in the correct position. Additionally, if you place any items below this table section, you must position them using relative positioning so that they move down as the user adds new rows.

Note: *Relative positioning must be activated in two places: in the Designer's Tools/Preferences and within the Arrange menu.*

As in duplicating pages, it is usually best to use an Add Row button to provide the user with a way to add additional rows. If you place this button below the template row within the form's build order, you

can use it as a static item in the build order, allowing you to place newly duplicated items before it in the build order.

Other ways of triggering row duplication include adding a row every time the user leaves the last item of the last row. This method guarantees that there will always be an empty row.

The example form *Item_Duplication.xfd* shows a table structure in action. You can open the form in the Designer or a text editor and examine the Add button to see the duplication computes.

Duplicating Items with the XML Model

If you are using the XML Data Model in your form, duplicating items become a little more challenging. When duplicating items that are bound to a particular instance, you must also duplicate the corresponding XML instance fragment and the matching binds for those items.

This is one of the more complex features you would ever add to a form, but really shows the power of XFDL and the XML Model.

To start you will create the items that you wish to duplicate, as in the "Duplicating Items" section. You will then create your XML Instance and Binds for these items. Once this is complete you will take a copy of the XML Instance you created and make that your template when duplicating rows. You will also do the same with the Binds that you have created.

A simple example of this would be a form that captures customer information, with a row you plan to duplicate; given, middle, and last name.

Each row will contain a delete button and there will be a add button after the last row on the form. There will also be a hidden template page that contains the customer information item; text boxes, fields, and a delete button.

Examples

Below is the instance data used to hold the data from the three fields:

```
<xforms:instance xmlns="http://www.pureedge.com/PECustomerService"
id="MAIN_DATA">
   <company_XYZ>
      <customers>
         <customer>
            <given_name></given_name>
            <middle_name></middle_name>
            <last_name></last_name>
         </customer>
      </customers>
   </company_XYZ>
</xforms:instance>
```

Below are the "Binds" used to connect the data from the instance to the fields on the form.

```
<bind>
```

```
   <pecs:bind_row_count>1</pecs:bind_row_count>
   <instanceid>MAIN_DATA</instanceid>
   <ref>[pecs:company_XYZ][pecs:customers][0][pecs:given_name]</
ref>                  <boundoption>PAGE1.GNAME_FIELD_1.value</
boundoption>
</bind>
<bind>
   <pecs:bind_row_count>1</pecs:bind_row_count>
   <instanceid>MAIN_DATA</instanceid>
   <ref>[pecs:company_XYZ][pecs:customers][0][pecs:middle_name]</
ref>
   <boundoption>PAGE1.MNAME_FIELD_1.value</boundoption>
</bind>
<bind>
   <pecs:bind_row_count>1</pecs:bind_row_count>
   <instanceid>MAIN_DATA</instanceid>
   <ref>[pecs:company_XYZ][pecs:customers][0][pecs:last_name]</
ref>                  <boundoption>PAGE1.LNAME_FIELD_1.value</
boundoption>
</bind>
```

Next we need to set up our template Instance and Binds.

Template Instance:

```
<xforms:instance xmlns="http://www.pureedge.com/PECustomerService"
id="TEMPLATE_INSTANCE">
   <TEMPLATE>
      <customers>
         <customer>
            <given_name></given_name>
            <middle_name></middle_name>
            <last_name></last_name>
         </customer>
      </customers>
   </TEMPLATE>
</xforms:instance>
```

The instance above is the same as our previous instance, except it is nested in the "TEMPLATE" tag and given a different id of "TEMPLATE_INSTANCE" so that they can be distinguished from each other.

Template Binds:

```
<pecs:template_binds>
   <bind>
      <pecs:bind_row_count>0</pecs:bind_row_count>
      <instanceid>MAIN_DATA</instanceid>
      <ref compute="'[pecs:company_XYZ][pecs:customers]     &#xA;
         [' +. (..[pecs:bind_row_count] - '1') +. ']        &#xA;
```

```
            [pecs:given_name]'"
            >[pecs:company_XYZ][pecs:customers][-
1][pecs:given_name]</ref>
         <boundoption compute="'PAGE1.GNAME_FIELD_' +.         &#xA;
            (..[pecs:bind_row_count]) +.                       &#xA;
            '.value'">PAGE1.GNAME_FIELD_0.value</boundoption>
      </bind>
      <bind>
         <pecs:bind_row_count>0</pecs:bind_row_count>
         <instanceid>MAIN_DATA</instanceid>
         <ref compute="'[pecs:company_XYZ][pecs:customers][' +.&#xA;
            (..[pecs:bind_row_count] - '1') +. ']              &#xA;
            [pecs:middle_name]'"
            >[pecs:company_XYZ][pecs:customers][-
1][pecs:middle_name]</ref>
         <boundoption compute="'PAGE1.MNAME_FIELD_' +.         &#xA;
            (..[pecs:bind_row_count]) +. '.value'"
            >AGE1.MNAME_FIELD_0.value</boundoption>
      </bind>
      <bind>
         <pecs:bind_row_count>0</pecs:bind_row_count>
         <instanceid>MAIN_DATA</instanceid>
         <ref compute="'[pecs:company_XYZ][pecs:customers][' +.&#xA;
            (..[pecs:bind_row_count] - '1') +. '][pecs:last_name]'"
            >[pecs:company_XYZ][pecs:customers][-
1][pecs:last_name]</ref>
         <boundoption compute="'PAGE1.LNAME_FIELD_' +.         &#xA;
         (..[pecs:bind_row_count]) +. '.value'"
         >PAGE1.LNAME_FIELD_0.value</boundoption>
      </bind>
   </pecs:template_binds>
```

The Binds above are the most complex part of this feature, because of their dynamic nature. Unlike the other binds that exist in the XFDL namespace these binds exist in a custom namespace that we have named "PECS".

Each of the computes found in these binds are being used to generate all of the references based on the "pecs:bind_row_count" variable. When it is time to duplicate the binds you supply this variable with the row number you are on and then the dynamic references get updated to the correct values.

Add Button:

This button creates the requested instance, binds, and items. First we duplicate all the items. We then duplicate the instance, as we want to assure the correct xml structure in the instance. The binds will create the instance for us if we choose, but the structure will be incorrect as all numerical references will be created as array elements (ex. <XFDL:ae>). Finally the binds are duplicated from the template and xmlmodelUpdate() is called to update everything.

```
   <pecs:duplicate_customer_information xfdl:compute="         &#xA;
```

```
   toggle(activated, 'off', 'on') == '1'                    &#xA;
   ? set('pecs:total_row_count', pecs:total_row_count + '1')&#xA;
   +. duplicate('TEMPLATE.GNAME_LABEL_1', 'item',
'ADDCUST_BUTTON', &#xA;
      'item', 'before_sibling',   'GNAME_LABEL_' +.         &#xA;
      pecs:total_row_count)                                 &#xA;
   +. duplicate('TEMPLATE.GNAME_FIELD_1', 'item',
'ADDCUST_BUTTON', &#xA;
      'item', 'before_sibling', 'GNAME_FIELD_' +.           &#xA;
      pecs:total_row_count)                                 &#xA;
   +. duplicate('TEMPLATE.MNAME_LABEL_1', 'item',
'ADDCUST_BUTTON', &#xA;
      'item', 'before_sibling', 'MNAME_LABEL_' +.           &#xA;
      pecs:total_row_count)                                 &#xA;
   +. duplicate('TEMPLATE.MNAME_FIELD_1', 'item',
'ADDCUST_BUTTON', &#xA;
      'item', 'before_sibling', 'MNAME_FIELD_' +.           &#xA;
      pecs:total_row_count)                                 &#xA;
   +. duplicate('TEMPLATE.LNAME_LABEL_1', 'item',
'ADDCUST_BUTTON', &#xA;
      'item', 'before_sibling', 'LNAME_LABEL_' +.           &#xA;
      pecs:total_row_count)                                 &#xA;
   +. duplicate('TEMPLATE.LNAME_FIELD_1', 'item',
'ADDCUST_BUTTON', &#xA;
      'item', 'before_sibling', 'LNAME_FIELD_' +.           &#xA;
      pecs:total_row_count)                                 &#xA;
   +. set(duplicate('TEMPLATE.DELCUST_BUTTON_1', 'item',    &#xA;
      'ADDCUST_BUTTON', 'item', 'before_sibling',
'DELCUST_BUTTON_' &#xA;
      +. pecs:total_row_count) +. '.pecs:row_count',        &#xA;
      pecs:total_row_count)                                 &#xA;
   +. duplicate('global.global.xmlmodel[instances][1]       &#xA;
      [pecs:TEMPLATE][pecs:customers][pecs:customer]', 'option',
&#xA;
      'global.global.xmlmodel[instances][0][pecs:company_XYZ]&#xA;
      [pecs:customers]', 'option', 'append_child',
'pecs:customer') &#xA;
   +. set(duplicate('global.global.xmlmodel[bindings]       &#xA;
      [pecs:template_binds][0]', 'option',                  &#xA;
      'global.global.xmlmodel[bindings][pecs:template_binds]',
&#xA;
      'option', 'BEFORE_SIBLING', 'bind') +.                &#xA;
      '[pecs:bind_row_count]', pecs:total_row_count)        &#xA;
   +. set(duplicate('global.global.xmlmodel[bindings]       &#xA;
      [pecs:template_binds][1]', 'option',                  &#xA;
      'global.global.xmlmodel[bindings][pecs:template_binds]',
&#xA;
      'option', 'BEFORE_SIBLING', 'bind') +.                &#xA;
      '[pecs:bind_row_count]', pecs:total_row_count)        &#xA;
```

```
   +. set(duplicate('global.global.xmlmodel[bindings]       &#xA;
      [pecs:template_binds][2]', 'option',                  &#xA;
      'global.global.xmlmodel[bindings][pecs:template_binds]',
&#xA;
      'option', 'BEFORE_SIBLING', 'bind') +.                &#xA;
      '[pecs:bind_row_count]', pecs:total_row_count)        &#xA;
   +. xmlmodelUpdate(): ''"></pecs:duplicate_customer_information>
<pecs:total_row_count>1</pecs:total_row_count>
```

Delete Button and Action:

The Delete Button calls the action to destroy the row. This is performed in an action because we destroy the delete button, and destroy is not allowed to destroy itself.

```
<pecs:destroy_customer_information xfdl:compute="           &#xA;
   toggle(activated, 'off', 'on') == '1'  ?                 &#xA;
   set('DEL_ROW_ACTION.pecs:row_to_delete',                 &#xA;
   ADDCUST_BUTTON.pecs:total_row_count)                     &#xA;
   +. set('DEL_ROW_ACTION.pecs:instance_to_delete',         &#xA;
   pecs:row_count - '1') +. set('DEL_ROW_ACTION.active', 'on') :
&#xA;
   ''"></pecs:destroy_customer_information>
```

This function destroys the requested row and the instance and binds to go with that row. To do this we destroy the last row and binds created and then set delete the requested instance data. We do it this way, so there is no need to modify any of the row_counts within the binds and when deleting the appropriate instance data will shuffle accordingly when xmlmodelUpdate() is called.

```
<pecs:delete_Requested xfdl:compute="                       &#xA;
   toggle(activated, 'off', 'on') == '1' ?                  &#xA;
   set('pecs:template_bindings_index',                      &#xA;
      getPosition('global.global.xmlmodel[bindings]         &#xA;
      [pecs:template_binds]', 'array', 'XFDL'))             &#xA;
   +. destroy('global.global.xmlmodel[bindings][' +.        &#xA;
      (pecs:template_bindings_index - '3') +. ']', 'array') &#xA;
   +. destroy('global.global.xmlmodel[bindings][' +.        &#xA;
      (pecs:template_bindings_index - '3') +. ']', 'array') &#xA;
   +. destroy('global.global.xmlmodel[bindings][' +.        &#xA;
      (pecs:template_bindings_index - '3') +. ']', 'array') &#xA;
   +. destroy('global.global.xmlmodel[instances][0]         &#xA;
      [pecs:company_XYZ][pecs:customers][' +.               &#xA;
      (pecs:instance_to_delete) +. ']' ,  'array')          &#xA;
   +. destroy('GNAME_LABEL_' +. pecs:row_to_delete, 'item') &#xA;
   +. destroy('GNAME_FIELD_' +. pecs:row_to_delete, 'item') &#xA;
   +. destroy('MNAME_LABEL_' +. pecs:row_to_delete, 'item') &#xA;
   +. destroy('MNAME_FIELD_' +. pecs:row_to_delete, 'item') &#xA;
   +. destroy('LNAME_LABEL_' +. pecs:row_to_delete, 'item') &#xA;
   +. destroy('LNAME_FIELD_' +. pecs:row_to_delete, 'item') &#xA;
```

```
+. destroy('DELCUST_BUTTON_'+. pecs:row_to_delete, 'item')&#xA;
+. xmlmodelUpdate()                                          &#xA;
+. set('ADDCUST_BUTTON.pecs:total_row_count',               &#xA;
   ADDCUST_BUTTON.pecs:total_row_count - '1')               &#xA;
+. set('active', 'off') : ''"></pecs:delete_Requested>
```

Further Examples

You can refer to the *Dynamic_Instance_Bind_Simple_Example.xfd* form for an additional example. This form is best viewed in the Form View window of Workplace Forms Designer or within a text editor. While in this view mode look for the string, "COMMENT" to find all the computes that pertain to this example and explanations for compute section.

Deleting Items and Pages

You can use the destroy function to delete an entire page or and individual item. This allows the user to remove an unknown or infinite number of pages and items from the form.

Destroy deletes specified elements from a form, including boxes, buttons, checks, comboboxes, fields, labels, lines, lists, pages, popups, radios, and spacers.

When a compute within the destroy function causes a change to the form, the form checks for other computes referenced by the destroy function and evaluates them immediately.

The following example shows a button designed to give the form user the option to delete an optional comments page:

```
<button sid=”remove_comments”>
   <value>Remove Comments Page</value>
   <custom:remove_page xfdl:compute = "toggle(activated,    &#xA;
   'off', 'on') == '1' ? destroy('comments_page', 'page')   &#xA;
   : ''"></custom:remove_page>
</button>
```

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user' responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION " IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation Office

4360 One Rogers Street

Cambridge, MA 02142

U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the

same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks

IBM, the IBM logo, Workplace Forms, DB2, and WebSphere are trademarks of International Business Machines Corporation in the United States, other countries, or both:

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.